Mathematics of Rotating Armour Stands

Sheepe

August 11, 2021

Contents

1 Problem **2**

2 Prerequisites **2**

3 Method **3**

4 Converting a Ray to Object Space **4**

5 Finding the CFrame of an Armour Stand **5**

6 Intersecting the Ray with a Body Part **6**
6.1 Rotation of Limbs with ZYX Euler Angles 6
6.2 Finding a body part's CFrame matrix 6

7 Interaction Solution 1: Rotating on a plane **8**
7.1 Finding the Axis Angle 8
7.2 Finding the Angle 9
7.2.1 Finding the sine and cosine of the angle 9
7.2.2 Finding the player's intersection with the plane 10
7.3 Finding the new orientation of the arm 11

8 Interaction Solution 2: Point selection on a sphere **11**
8.1 Details of solving 11
8.2 Ray-Sphere Intersection 12

1 Problem

The problem in question is to be able to do the following:

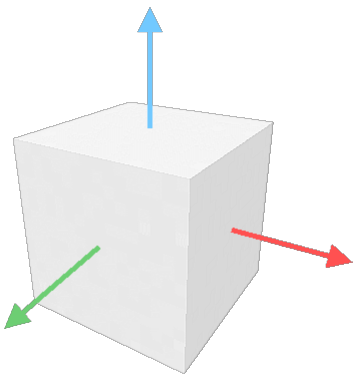
1. Detect when a player looks at a body part of an armour stand
2. Allow the player to drag the body parts by some mechanism.

we will go through the process doing this mathematics and explore some ideas.

2 Prerequisites

The main piece of mathematics used here is based on ROBLOX's CFrame object that is used for manipulating the 3d rotations and positions of objects at the same time. The way this is stored / calculated is using a 4×4 matrix.

Firstly, we require the rotation matrix of the object, which can be figured out column by column. We need to find three vectors, an **up vector**, a **right vector** and a **look vector**. These can be figured out easily by taking a look at a rotated cube. *Note that these must be unit vectors.*



The normal of the face toward you is the *look vector* (green), the face to the right of that is the *right vector* (red), and the normal of the face on top is the *up vector* (blue).

This will be important for getting into what is called an object space. Essentially, these three vectors become the x, y and z axes for the 3D space relative to this object, hence is called it's object space. Using the standard 3D axes, we have what is called world space.

Once we have these three vectors, we can form the rotation matrix as they form the columns of the matrix, in the order right, up and then look vector. So if we have the rotation matrix:

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

then we have the up vector (m_{11}, m_{21}, m_{31}), right vector (m_{12}, m_{22}, m_{32}) and look vector (m_{13}, m_{23}, m_{33}). I did say that CFrames also handle positioning, so how is that? Well, we take the x, y, z coordinates in world space of the object, and we form the following matrix:

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & x \\ m_{21} & m_{22} & m_{23} & y \\ m_{31} & m_{32} & m_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

this now encodes both the position and the rotation of an object, and by multiplying these matrices together, we can translate and rotate in object space.

It should be noted that these objects should not be added / subtracted in a standard way. When we add/subtract CFrames we are referring only to the position element of the CFrame and not the rotation.

For any CFrame C, we will denote $\hat{C}$ the CFrame in which the rotation has been removed, i.e, replaced by the identity matrix, and any vector with a subscript o will mean that vector in the contextually-relevant object space, and finally for any vector $w = (w_x, w_y, w_z)$, we denote $\overline{w} = (w_x, w_y, w_z, 1)$. You will see why this is relevant later.

3 Method

To solve this problem (1), we have to transform between a few different object spaces. The method I am about to describe will work for any particular limb / body part of the armour stand.

To detect if a ray goes through an axis aligned bounding box in world space is fairly simple. The problem we are trying to tackle is to detect if a ray in world space goes through a rotated bounding box in world space. There is a simple way to solve this problem:

1. Find the coordinates of the ray in the object space of the bounding box instead of world space.

2. Perform the intersection detection in object space.

Whilst we only know how to perform the intersection in world space, world space isn't actually "special", it's just a standard we use. We just have to make sure that any coordinates we use with it are actually in object space.

To solve problem (2), we will explore two different methods.

1. Intersect the ray with a sphere and rotate the arm to the closest intersection point.

2. Put a plane at the pivot with a normal facing the player directly, and rotate in the plane.

4 Converting a Ray to Object Space

For this section, assume the CFrame defining the object space is denoted by C, an arbitrary CFrame.

Suppose that we have a ray, (v, d), where $v = (v_x, v_y, v_z)$ is the point in 3D space which defines where the ray is, and $d = (d_x, d_y, d_z)$ is the direction of the ray (unit).

To get the transformed point, we multiply the inverse CFrame by $\overline{v}$:

$$\overline{v_o} = C^{-1}\overline{v}$$

This makes sense, as multiplying C by the point directly would be converting the point from object space to world space. This is because it applies the translation and rotation to the point. We want to do the opposite - remove the translation and rotation, so we want the inverse matrix.

To transform the vector / direction, we can't do this directly. When we multiply by the inverse CFrame, it rotates and translates. We don't want the vector to be translated, only rotated. So we do the following operation:

$$\overline{d_o} = (C^{-1} - \hat{C^{-1}})\overline{v}$$

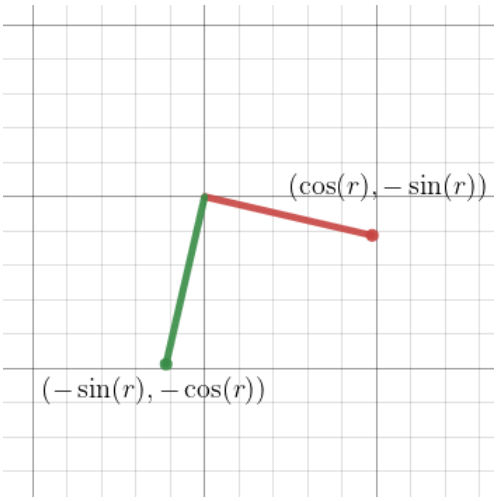
Notice that $C^{-1} - \hat{C^{-1}}$ will be a CFrame with no position and only the rotation of C^{-1}, and then we are rotating the vector $\overline{v}$, converting it into object space.

Hence the final converted ray is $(\overline{v_0}, \overline{d_o}) = (C^{-1}\overline{v}, C^{-1} - \hat{C^{-1}})\overline{v})$.

5 Finding the CFrame of an Armour Stand

A way we can simplify our situation here is that whilst Armour Stands have two rotation components `[0.0f, 0.0f]` the second one has no visual impact. So from henceforth we will describe the rotation of the entire armour stand as simply the first coordinate here, and will generally be referred to as r.

So, if this is the only rotation that matters, how can we get our rotation matrix for the armour stand? The up vector is straight forward, just being $(0, 1, 0)$. For the right vector and look vector, we can look at the problem in 2D and use some basic sin and cos.



Using this, we can get the CFrame of the armour stand pretty simply, defining $c_r = \cos(r)$, $s_r = \sin(r)$ and assuming that the position of the armour stand is (p_x, p_y, p_z). We need to make a small edit here however. For our calculations, we will want to be using the position that is right at the armour stand's feet, and in Minecraft the armour stand's position is below the base - hence, we must add a pixel height onto the y-coordinate of $\frac{1}{16}$. This yields the following matrix:

$$A = \begin{bmatrix} c_r & 0 & -s_r & p_x \\ 0 & 1 & 0 & p_y + \frac{1}{16} \\ -s_r & 0 & -c_r & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and note that we're denoting the CFrame of any particular armour stand as A.

6 Intersecting the Ray with a Body Part

6.1 Rotation of Limbs with ZYX Euler Angles

This process will work for any body part, so without loss of generality we will perform this process on the right arm. We will be finding the CFrame of the right arm in this case. This means, as in the prerequisites, we will need to find the rotation matrix for the arm. The position can be calculated as you will see soon.

Rotations in Minecraft are handled with Euler Angles. This means that a rotation on each axis is applied in a specific order. In our case, Z, then Y, then X. We can find the rotation of any particular limb (in this case, right arm) from the pose NBT, and we can use this to calculate the matrix.

We will denote: c and s to be cos/sin respectively of the angle in their subscript, so for example c_x is the cosine of the x rotation. *However, due to weirdness with the rotations both the y and z angles should be negated. This has already been done in the following matrix.*

$$\begin{bmatrix} c_z c_y & c_z s_y s_x - c_x s_z & -s_z s_x - c_z c_x s_y \\ c_y s_z & c_z c_x + s_z s_y s_x & -c_x s_z s_y + c_z s_x \\ s_y & -c_y s_x & c_y c_x \end{bmatrix}$$

This looks very complicated, however it really is just the multiplication of three rotation matrices on three orthogonal axes in the order ZYX with the angles changed as was described above.

So, we can indeed use this to define the rotation of any limb, but for our calculations, it needs to be a CFrame so we will define:

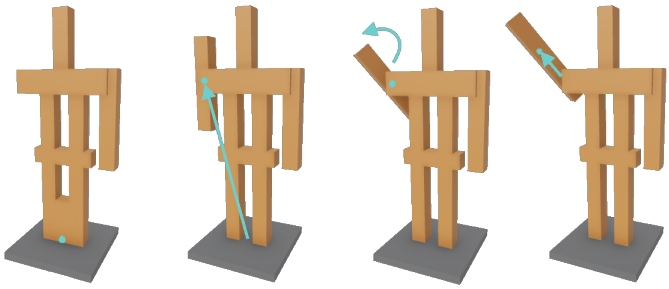
$$B = \begin{bmatrix} c_z c_y & c_z s_y s_x - c_x s_z & -s_z s_x - c_z c_x s_y & 0 \\ c_y s_z & c_z c_x + s_z s_y s_x & -c_x s_z s_y + c_z s_x & 0 \\ s_y & -c_y s_x & c_y c_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6.2 Finding a body part's CFrame matrix

So, how do we get the limbs position with this? Well first we have to address that the limbs do not rotate around their geometric centres, but instead at pivot points fixed with respect to the armour stand. Hence, we must work in the object space of the armour stand.

To rotate around a pivot point, we first translate object space to the pivot, rotate it to the desired rotation, and then we translate the object in it's object space to align pivot to the correct spot on the object.

This sounds a bit complicated, so the diagram below goes through the process step by step:



We move to the body's object space, then we translate to the object space of the pivot, then we rotate in the object space of the pivot, and then we translate in the object space of the newly rotated arm.

To write these calculations in a clean way, we will first define the following:

$$P_{x,y,z} = \begin{bmatrix} 1 & 0 & 0 & \frac{x}{16} \\ 0 & 1 & 0 & \frac{y}{16} \\ 0 & 0 & 1 & \frac{z}{16} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which creates a non-rotated CFrame at a particular coordinate. Since we will be using this relative to the armour stand, we are dividing by 16 so we provide texture pixel measurements instead of actual block distances for convenience.

Finally, we can do the math. To move into the object space of the armour stand, we simply start with the CFrame A.

In this particular case, in this object space we need to count the number of pixels left, up and back that the pivot point is for the right arm, and this happens to be $(-6, 21, 0)$. So to create a CFrame for the second image, we simply just have $AP_{-6,21,0}$.

For the third picture, we now multiply by the rotation CFrame that we found earlier, B, giving us $AP_{-6,21,0}B$.

For the last picture, how do we find this translation? Well imagine that it was never rotated in the third picture. What translation is required to re-align the arm to it's original position? In the case of the right arm, it happens to be $(0, -4, 0)$.

This gives us a final CFrame of $AP_{-6,21,0}BP_{0,-4,0}$. Now you ask me, how does this help us intersect it? Well, we've now got the object space of the limb. This means that we can find the ray in this object space, but also find the coordinates of two opposite corners of the bounding box **in the object space**. This is trivial as the axes are long the normals of the box and the centre of the box is the origin of the space.

To find this, we again imagine that the limb has not been rotated at all, and we count the pixels from the centre of the limb to any two completely opposite corners of the box and divide by 16. For the right or left arm, one option is $(-1/16, 6/16, -1/16)$ and $(1/16, -6/16, 1/16)$.

We can then use our ray-aabb intersection algorithm as usual with these new numbers and there we go! We can see whether the player is hovering over any rotated or unrotated body part of the armour stand with their cursor with great accuracy.

7 Interaction Solution 1: Rotating on a plane

7.1 Finding the Axis Angle

For this to work, we need to be a bit more specific. We aren't necessarily rotating on a plane, we are rotating around an axis. More technically, an axis is a unit vector that extends from the centre of the object, around which you rotate a certain angle.

To apply this to CFrames, we just need the rotation matrix. Luckily, smart people before us have already done the work. The matrix for rotating an angle θ around an axis $u = (u_x, u_y, u_z)$ where $c_\theta = \cos(\theta)$ and $s_\theta = \sin(\theta)$:

$$\begin{bmatrix} c_\theta + u_x^2(1-c_\theta) & u_x u_y(1-c_\theta) - u_z s_\theta & u_x u_z(1-c_\theta) + u_y s_\theta \\ u_y u_x(1-c_\theta) + u_z s_\theta & c_\theta + u_y^2(1-c_\theta) & u_y u_z(1-c_\theta) - u_x s_\theta \\ u_z u_x(1-c_\theta) - u_y s_\theta & u_z u_y(1-c_\theta) + u_x s_\theta & c_\theta + u_z^2(1-c_\theta) \end{bmatrix}$$

Yes, I know. It's disgusting. We can make a CFrame from this by just having no translation components, and define:

$$R_{u,\theta} = \begin{bmatrix} c_\theta + u_x^2(1-c_\theta) & u_x u_y(1-c_\theta) - u_z s_\theta & u_x u_z(1-c_\theta) + u_y s_\theta & 0 \\ u_y u_x(1-c_\theta) + u_z s_\theta & c_\theta + u_y^2(1-c_\theta) & u_y u_z(1-c_\theta) - u_x s_\theta & 0 \\ u_z u_x(1-c_\theta) - u_y s_\theta & u_z u_y(1-c_\theta) + u_x s_\theta & c_\theta + u_z^2(1-c_\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So how do we actually apply this? Well we first have to consider what the axis is. If we want the player to stand still and rotate the arm around the axis, we need to simply get the vector from the pivot point to the players

head.

Luckily, we can get this information with relative ease. Back in section 6, the CFrame we have for the second image has the position, namely $AP_{-6,21,0}$. We just ignore the rotation components, and take the position vector from it and then subtract it from the player's head position and normalize.

In Minecraft, this can be done by placing an entity at the pivot, forcing it to face the player's head, moving forward one block locally and taking the difference between it's new position and the pivot position - and that's the axis! We will denote this unit vector axis as n.

7.2 Finding the Angle

7.2.1 Finding the sine and cosine of the angle

But you could now ask, okay, how do we know how far to rotate it? There are several problems here to solve. Firstly, the limb / body part is not guaranteed to lay within the plane, so we have to project the vector onto the plane.

To project a vector v onto the plane, we simply just need to subtract the projection of it onto the normal vector n from itself: $v - (v \cdot n)n$.

So, we need to get the vector along the length of the arm. To do this, we just need to take the pivot and any other point on the limb, for example, the centre of the object. We then do that position subtract the pivot and that is our v. We have to infact normalize it, which is very easy in Minecraft. We just put an entity at the pivot, face it towards the second position and move it forward a block, subtract the new position from the pivot.

From here, our job is actually relatively simple. We just need to find the vector between the pivot and where the user's looking direction hits the plane, and take the dot product of it with the projected vector:

$$v \cdot u = |v||u| \cos(\theta)$$

This will give us the cosine of the angle between the two vectors, as $|u| = |v| = 1$. We can use this in the axis angle matrix straight away. *Note that if v is not normalized, this won't be true and it will have been multiplied by the length of v which is much harder to compute.*

But don't we also need the sine of the angle as well? Yes we do! Fortunately, there's a nifty identity that can allow us to calculate sin, $\cos^2 x + \sin^2 x = 1$. However we don't have access to square roots in Minecraft - so we have to be a bit more creative.

Here, we actually will utilize the cross product of two vectors in 3D, given by the following formula:

$$v \times u = |v||u| \sin(\theta) n$$

We would actually cross the two vectors in that lay in the plane, that is, the normalized projected vector of the limb and the unit vector made from the user's mouse intersecting the plane. Then, n would be the normal of the plane that we already computed before.

Since v and u are both unit vectors, we can actually say that:

$$v \times u = n \sin(\theta)$$

Both sides are vectors, so their components must be equivalent. Since n is a unit vector, at least one of n_x, n_y, n_z must be nonzero, so we just take the corresponding cross product component and divide it by the normal vector n non-zero component, and there we go! We have the sine of the angle between the vectors.

7.2.2 Finding the player's intersection with the plane

For the player's looking ray, (p, d) where p is the player's head position and d is the unit looking direction of the player (to find this, move a block forward of the players head and subtract the new position from the head position), we first need to define an important metric. We will also define u to be the pivot of the relevant limb.

If $n \cdot d$ is zero, that means that there will be no intersection into the plane, so we may want to limit this to some threshold above zero. Otherwise, we can define every point on the ray as $r = p + dt$, then:

$$t = \frac{(u - p) \cdot n}{n \cdot d}$$

(We can also see here why $n \cdot d$ cannot be zero). Using this, we can easily find the intersection point on the plane, it's simply $p + dt$. So, our vector in the plane that is used for the other calculations is $p - u + dt$, we just need to normalize it first. Methods for normalizing vectors have already been discussed in this document for Minecraft.

So with all of this, we can find a rotation matrix which converts the limb CFrame into one which is more relevant to the players looking position, so we are ready for the next step.

7.3 Finding the new orientation of the arm

So, we have the "theoretical" CFrame of the arm, so what. We don't know the angles to set the armour stand to. This is actually quite difficult to find and will require approximations of inverse trigonometric functions[1]. Refer to the footnote for more information here, some axes may need to be negated, there's no way to know until the math has actually been done in-game.

This actually finishes the discussion for the first solution, as this is actually all that is needed. We form the plane with the normal connecting the pivot and the player's head, at the pivot. Then, we project the limb's internal vector onto the plane, and we find the vector of where the player wants the arm to go, on the plane. We calculate the angle, calculate the relevant rotation matrix and transform the limb CFrame. We then use reverse euler angles to calculate the new orientation.

There may be a simpler method to get to the final calculation but the general process will likely be the same or very similar.

8 Interaction Solution 2: Point selection on a sphere

8.1 Details of solving

Intersecting a ray with a sphere gives either 0, 1 or 2 points. 0 points and 1 point is easy to handle; for 0 points, we do nothing. For 1 point, we aim to make the limb point toward it. But what do we do for two points?

We can actually utilize the dot product here to find which of the intersection points is closest to the current angle the limb is at. This is because the dot product gives us a measure of how close two vectors are to each other, and when it's unit vectors, it's even easier.

If we denote the two intersection points p_1 and p_2 on a sphere centered at the pivot point, and l to be the unit vector representing the arm direction we initially referenced in the first solution, then we have the following condition:

$$\text{If } p_1 \cdot l > p_2 \cdot l, \text{ then } p_1 \text{ is a tighter angle towards } l \text{ than } p_2.$$

[1] https://www.geometrictools.com/Documentation/EulerAngles.pdf This page details psudeocode on how to find these angles. You will notice there are two edge cases, and when choosing which angle in these cases requires detecting which angle is closer on the sphere.

and of course the reverse is true. So we simply just choose the point which when dotted with l, has the higher dot product.

As for actually calculating the rotations, we do the exact same thing as in the first solution. We dot and cross the limb vector and the chosen vector for the point intersection on the sphere to get the sin and cos of their separating angle, and we actually rotate around the axis that is the vector that is the cross product of those two vectors.

Then we do the exact same as in the first solution - we rotate the CFrame, find the euler angles and apply to the armour stand.

8.2 Ray-Sphere Intersection

As in the previous section, we define l to be the unit vector representing the arm's direction. We will define o to be the pivot position in world space, and as usual, (p, d) to be the player's ray. The sphere's radius will be 1 as we would like to use unit vectors for everything.

Going through some algebra for the sphere with the ray having equation $r = p + dt$, we find that t satisfies the following equation:

$$(d \cdot d)t^2 + 2d \cdot (p - o)t + (p - o) \cdot (p - o) - 1 = 0$$

so defining $a = d \cdot d$, $b = 2d \cdot (p - o)$, $c = (p - o) \cdot (p - o) - 1$, we can look at the discriminant of this quadratic, and we have the following conditions:

1. If $b^2 - 4ac < 0$, the ray does not hit the sphere.
2. If $b^2 - 4ac = 0$, the ray hits the sphere at one point.
3. If $b^2 - 4ac > 0$, the ray intersects the sphere twice.

Using the quadratic formula, we can find the relevant values of t and use $r = p + dt$ to find values p_1 and p_2 (if it exists), and then do as the previous subsection detailed.

This document has been rather convoluted but it definitely covers most of the maths, apart from the inverse trigonometric function approximations, for doing the proposed problem. If you have any questions, direct them to myself.